Writing Apache 2.0 Modules

By Ryan Bloom, Senior Software Engineer, Covalent Technologies

One of the reasons Apache is such a popular server is that it is easy for people to extend Apache by writing modules. With the upcoming release of Apache 2.0, much of the module structure has changed, and modules will need to be re-written to work with 2.0. However, because the Apache developers also need to port modules, we have tried to make this move easy. This talk will discuss writing modules for 2.0, and hopefully will also help people in porting their existing modules.

Apache Module Types

Apache 2.0 has added to the types of modules that can be written. In addition to the standard modules that have been in Apache since version 0.8.8, Apache 2.0 also includes Multi-processing modules (MPMs) and Protocol modules. Standard modules are used to add functionality to the web server. A basic Apache installation can only serve static HTML pages without any authentication. Modules are used to extend the range of functions available to Apache.

MPMs determine how the web server maps requests to threads and processes. MPMs are a part of Apache's effort to perform well on all platforms. They provide a method for people intimate with a given platform to define the profile of the Apache server.

Protocol modules are used to add protocols to Apache. While developing Apache 2.0, the programers realized that much of the work done in Apache was applicable to many Internet servers. Once this realization was made, Apache 2.0 was modified slightly to allow for protocol modules. This allows programers to take advantage of Apache's framework with any protocol. One of the hardest parts of writing a server is to make the server reliable ad robust. Apache has already done this work, so it makes sense for other developers to take advantage of this work. Protocol modules allow developers to drop a new protocol into Apache. Currently Apache 2.0 has two protocol modules, HTTP and echo. The HTTP protocol module is is src/main, and is controlled by http_core. The echo module is in src/modules/standard, and is called mod_echo. This protocol listens to a port, and when a request comes in, it reads from the socket and echos the data read back to the client.

Although these module types are available, this talk will not discuss them in any detail. The lessons learned from writing standard modules can be applied to modules of the other types.

The easiest way to learn to write either MPM or Protocol modules, is to read a current module, and learn from it.

Apache Structures

There are a few structures that must be understood when writing Apache modules. The first structure is the server_rec. Apache understands about multiple virtual hosts in the same server, and when Apache is started, it creates on server_rec for each virtual host. Then, when a request is accepted by the server, the correct server_rec is attached to the request_rec (discussed below). The server_rec defines everything about the current server, such as the name, the port, the administrators contact information, etc. The full server_rec can be seen below. Some of the fields in this structure will be discussed in detail later in this talk.

```
struct server_rec {
    /** The process this server is running in */
    process_rec *process;
    /** The next server in the list */
    server_rec *next;

    /** The name of the server */
    const char *defn_name;
    /** The line of the config file that the server was defined on */
    unsigned defn_line_number;

    /* Contact information */

    /** The admin's contact information */
    char *server_admin;
    /** The server hostname */
    char *server_hostname;
    /** for redirects, etc. */
    unsigned short port;

    /* Log files --- note that transfer log is now in the modules... */

    /** The name of the error log */
    char *error_fname;
    /** A file descriptor that references the error log */
    apr_file_t *error_log;
    /** The log level for this server */
    int loglevel;

    /* Module-specific configuration for server, and defaults... */

    /** true if this is the virtual server */
    int is_virtual;
    /** Config vector containing pointers to modules' per-server config
     *  structures. */
    void *module_config;
    /** MIME type info, etc., before we start checking per-directory info */
    void *lookup_defaults;

    /* Transaction handling */

    /** I haven't got a clue */
    server_addr_rec *addrs;
    /** Timeout, in seconds, before we give up */
    int timeout;
    /** Seconds we'll wait for another request */
    int keep_alive_timeout;
    /** Maximum requests per connection */
    int keep_alive_max;
    /** Use persistent connections? */
    int keep_alive;

    /** Pathname for ServerPath */
    const char *path;
    /** Length of path */
    int pathlen;

    /** Normal names for ServerAlias servers */
    apr_array_header_t *names;
    /** Wildcarded names for ServerAlias servers */
    apr_array_header_t *wild_names;

    /** effective user id when calling exec wrapper */
    uid_t server_uid;
    /** effective group id when calling exec wrapper */
    gid_t server_gid;
```

```
    /** limit on size of the HTTP request line    */
    int limit_req_line;
    /** limit on size of any request header field */
    int limit_req_fieldsize;
    /** limit on number of request header fields  */
    int limit_req_fields;
};
```

The second structure involved in Apache is the conn_rec. This structure stores information about the current connection to Apache. It is possible to pipeline multiple requests over the same connection, so this structure stores information about the current connection. One of the fields in this structure is a BUFF structure. This is likely to go away in the future as a part of the filtering design. In Apache 1.3, the BUFF was used to store information before it was sent to the client. When the filtering implementation is finished, this structure may go away. However, modules should never notice this, because no module should be accessing the internals of the BUFF structure.

```
struct conn_rec {
    /** Pool associated with this connection */
    apr_pool_t *pool;
    /** Physical vhost this conn come in on */
    server_rec *base_server;
    /** used by http_vhost.c */
    void *vhost_lookup_data;

    /* Information about the connection itself */

    /** Connection to the client */
    BUFF *client;

    /* Who is the client? */

    /** local address */
    struct sockaddr_in local_addr;
    /** remote address */
    struct sockaddr_in remote_addr;
    /** Client's IP address */
    char *remote_ip;
    /** Client's DNS name, if known.  NULL if DNS hasn't been checked,
     *  "" if it has and no address was found.  N.B. Only access this though
     * get_remote_host() */
    char *remote_host;
    /** Only ever set if doing rfc1413 lookups.  N.B. Only access this through
     *  get_remote_logname() */
    char *remote_logname;

    /** Are we still talking? */
    unsigned aborted:1;
    /** Are we using HTTP Keep-Alive?  -1 fatal error, 0 undecided, 1 yes */
    signed int keepalive:2;
    /** Did we use HTTP Keep-Alive? */
    unsigned keptalive:1;
    /** have we done double-reverse DNS? -1 yes/failure, 0 not yet,
     *  1 yes/success */
    signed int double_reverse:2;

    /** How many times have we used it? */
    int keepalives;
    /** server IP address */
    char *local_ip;
    /** used for ap_get_server_name when UseCanonicalName is set to DNS
     *  (ignores setting of HostnameLookups) */
    char *local_host;

    /** ID of this connection; unique at any point in time */
    long id;
    /** Notes on *this* connection */
    void *conn_config;
    /** send note from one module to another, must remain valid for all
     *  requests on this conn */
    apr_table_t *notes;
};
```

The final structure is the request_rec. This could arguably be considered the most important structure for module writers. This structure stores information about the current request. This structure has a pointer to both the conn_rec and server_rec for this request. This allows modules to get information all possible information about the current request using just the request_rec. There is are three pointers to other request_recs. These give information about

the current request. If the next pointer is not NULL, then this request has been redirected. If the prev pointer is not NULL, then this request is an internal redirect, and if main is not NULL, then we are currently executing a sub request.

```
struct request_rec {
    /** The pool associated with the request */
    apr_pool_t *pool;
    /** The connection over which this connection has been read */
    conn_rec *connection;
    /** The virtual host this request is for */
    server_rec *server;

    /** If we wind up getting redirected, pointer to the request we
     *  redirected to.  */
    request_rec *next;
    /** If this is an internal redirect, pointer to where we redirected
     *  *from*.  */
    request_rec *prev;

    /** If this is a sub_request (see request.h) pointer back to the
     *  main request.  */
    request_rec *main;

    /* Info about the request itself... we begin with stuff that only
     * protocol.c should ever touch...
     */
    /** First line of request, so we can log it */
    char *the_request;
    /** HTTP/0.9, "simple" request */
    int assbackwards;
    /** A proxy request (calculated during post_read_request/translate_name) */
    int proxyreq;
    /** HEAD request, as opposed to GET */
    int header_only;
    /** Protocol, as given to us, or HTTP/0.9 */
    char *protocol;
    /** Number version of protocol; 1.1 = 1001 */
    int proto_num;
    /** Host, as set by full URI or Host: */
    const char *hostname;

    /** When the request started */
    apr_time_t request_time;

    /** Status line, if set by script */
    const char *status_line;
    /** In any case */
    int status;

    /* Request method, two ways; also, protocol, etc..  Outside of protocol.c,
     * look, but don't touch.
     */

    /** GET, HEAD, POST, etc. */
    const char *method;
    /** M_GET, M_POST, etc. */
    int method_number;
    int allowed;                /* Allowed methods - for 405, OPTIONS, etc */
    apr_array_header_t *allowed_xmethods; /* Array of extension methods */

    /** byte count in stream is for body */
    int sent_bodyct;
    /** body byte count, for easy access */
    long bytes_sent;
    /** Time the resource was last modified */
    apr_time_t mtime;

    /* HTTP/1.1 connection-level features */

    /** sending chunked transfer-coding */
    int chunked;
    /** number of byte ranges */
    int byterange;
    /** multipart/byteranges boundary */
    char *boundary;
    /** The Range: header */
    const char *range;
    /** The "real" content length */
    long clength;

    /** bytes left to read */
    long remaining;
    /** bytes that have been read */
    long read_length;
    /** how the request body should be read */
```

```
    int read_body;
    /** reading chunked transfer-coding */
    int read_chunked;
    /** is client waiting for a 100 response? */
    unsigned expecting_100;
    /* MIME header environments, in and out.  Also, an array containing
     * environment variables to be passed to subprocesses, so people can
     * write modules to add to that environment.
     *
     * The difference between headers_out and err_headers_out is that the
     * latter are printed even on error, and persist across internal redirects
     * (so the headers printed for ErrorDocument handlers will have them).
     *
     * The 'notes' apr_table_t is for notes from one module to another, with no
     * other set purpose in mind...
     */

    /** MIME header environment from the request */
    apr_table_t *headers_in;
    /** MIME header environment for the response */
    apr_table_t *headers_out;
    /** MIME header environment for the response, printed even on errors and
     * persist across internal redirects */
    apr_table_t *err_headers_out;
    /** Array of environment variables to be used for sub processes */
    apr_table_t *subprocess_env;
    /** Notes from one module to another */
    apr_table_t *notes;

    /* content_type, handler, content_encoding, content_language, and all
     * content_languages MUST be lowercased strings.  They may be pointers
     * to static strings; they should not be modified in place.
     */
    /** The content-type for the current request */
    const char *content_type;   /* Break these out --- we dispatch on 'em */
    /** The handler string that we use to call a handler function */
    const char *handler;        /* What we *really* dispatch on           */

    /** How to encode the data */
    const char *content_encoding;
    /** for back-compat. only -- do not use */
    const char *content_language;
    /** array of (char*) representing the content languages */
    apr_array_header_t *content_languages;

    /** variant list validator (if negotiated) */
    char *vlist_validator;

    /** If an authentication check was made, this gets set to the user name. */
    char *user;
    /** If an authentication check was made, this gets set to the auth type. */
    char *ap_auth_type;

    /** This response is non-cache-able */
    int no_cache;
    /** There is no local copy of this response */
    int no_local_copy;
    /* What object is being requested (either directly, or via include
     * or content-negotiation mapping).
     */

    /** the uri without any parsing performed */
    char *unparsed_uri;
    /** the path portion of the URI */
    char *uri;
    /** The filename on disk that this response corresponds to */
    char *filename;
    /** The path_info for this request if there is any. */
    char *path_info;
    /** QUERY_ARGS, if any */
    char *args;
    /** ST_MODE set to zero if no such file */
    apr_finfo_t finfo;
    /** components of uri, dismantled */
    uri_components parsed_uri;

    /* Various other config info which may change with .htaccess files
     * These are config vectors, with one void* pointer for each module
     * (the thing pointed to being the module's business).
     */

    /** Options set in config files, etc. */
    void *per_dir_config;
    /** Notes on *this* request */
    void *request_config;

/**
```

```
 * a linked list of the configuration directives in the .htaccess files
 * accessed by this request.
 * N.B. always add to the head of the list, _never_ to the end.
 * that way, a sub request's list can (temporarily) point to a parent's list
 * @defvar const htaccess_result *htaccess
 */
    const struct htaccess_result *htaccess;

#ifdef APACHE_XLATE
    /** The translation headers for dealing with this request
     *  @defvar ap_rr_xlate *rrx */
    struct ap_rr_xlate *rrx;
#endif /*APACHE_XLATE*/

    /** A list of filters to be used for this request
     *  @defvar ap_filter_t *filters */
    struct ap_filter_t *filters;

/* Things placed at the end of the record to avoid breaking binary
 * compatibility.  It would be nice to remember to reorder the entire
 * record to improve 64bit alignment the next time we need to break
 * binary compatibility for some other reason.
 */
};
```

The Module Structure

The next structure that is important to Apache modules, is the module structure. As Apache executes, it calls out to modules at predefined locations. The module structure allows modules to register interest in being called at some of these locations. In Apache 1.3, all of the locations were specified in the module structure, this was an issue whenever a new hook was added to the structure, because all modules needed to be re-compiled. With Apache 2.0, this limitation was removed by removing most of the hooks from the module structure. Those hooks were replaced with a single function, the register_hooks function. Each module specifies a register_hooks function, which is then used to actually register interest in the remaining hooks for the server. Most modules do not use every function provided by the module structure and the hooks functions, but the example module has been written specifically to use them all, so this talk will refer to that module throughout the rest of this paper.

```
module example_module =
{
    STANDARD20_MODULE_STUFF,
    example_create_dir_config,    /* per-directory config creator */
    example_merge_dir_config,     /* dir config merger */
    example_create_server_config, /* server config creator */
    example_merge_server_config,  /* server config merger */
    example_cmds,                 /* command table */
    example_handlers,             /* list of content delivery handlers */
    example_register_hooks,       /* set up other request processing hooks */
};
```

The first field of the module structure is actually a macro that defines all of the information about the current module. This line should always be STANDARD20_MODULE_STUFF.

Directory Configuration

Apache can be configured with different properties for each directory. The second two fields in the module structure allow modules to be configured for each directory. The second field is the directory config creator. This function is called whenever a directory section is found in the config file. Modules should use this hook to setup defaults for the directory. The prototype for this function is:

```
static void *example_create_dir_config(apr_pool_t *p, char *dirspec);
```

The first argument is a pool to use for all memory allocation. The second is the name of the directory that is currently being configured. Modules should return a pointer to their

configuration structure. This structure can be retrieved later by calling ap_get_module_config with the per_dir_config field from the request_rec.

The second of these functions is the merger function. By default directories inherit the configuration options from their parent directories. If this is not the desired behavior, then modules need to implement a merger function. The prototype for this functions is:

```
static void *example_merge_dir_config(apr_pool_t *p, void *parent_conf,
                                      void *newloc_conf);
```

The module must not modify any of the structures passed into this function. Instead, modules should allocate space for the merged structure, and fill it out themselves.

Server Configuration

Besides configuring each directory, Apache allows modules to be configured for each server as well. This is done with two functions that mimic the directory behavior. The general ideas are the same, so I am not going to go into great detail here. The creator prototype is:

```
static void *example_create_server_config(apr_pool_t *p, server_rec *s);
```

The second argument is the server_rec for the current server. Again, modules should return a pointer to the server configuration structure. This can be retrieved by calling ap_get_module_config with the module_config field from the server_rec.

The second function is analogous to the dir_config_merger, and the prototype is:

```
static void *example_merge_server_config(apr_pool_t *p, void *server1_conf,
                                         void *server2_conf);
```

Again, modules should not modify any of the arguments passed to the function, and they return a pointer to the newly allocated server configuration structure.

Command Table

The fifth field of the module structure is the command table. Modules are allowed to implement their own directives that are valid in either the configuration file or .htaccess files. The command table is a table of command_rec structures. Directives are specified like the following:

```
{
    "Example",                          /* directive name */
    cmd_example,                        /* config action routine */
    NULL,                               /* argument to include in call */
    OR_OPTIONS,                         /* where available */
    NO_ARGS,                            /* arguments */
    "Example directive - no arguments"  /* directive description */
};
```

The final directive in the table must be NULL. That tells Apache that the module has no more directives to declare. The first field in the command_rec is the name of the directive. This is the name that will appear in the configuration file. The second field is the function to call when the directive is encountered. The third is a pointer to any information that should be passed to the function. Most directives use NULL for this field. The arguments from the configuration file are always passed to the function. The fourth field is where the directive is valid. It is the bitwise or value of:

```
        RSRC_CONF       --      .conf outside <Directory> or <Location>
        ACCESS_CONF     --      .conf inside <Directory> or <Location>
        OR_AUTHCFG      --      .conf inside <Directory> or <Location> and .htaccess
when                            AllowOverride AuthConfig
        OR_LIMIT        --      .conf inside <Directory> or <Location> and .htaccess
when                            AllowOverride Limit
        OR_OPTIONS      --      .conf anywhere and .htaccess when AllowOverride Options
        OR_FILEINFO     --      .conf anywhere and .htaccess when AllowOverride FileInfo
        OR_INDEXES      --      .conf anywhere and .htaccess when AllowOverride Indexes
        EXEC_ON_READ    --      Call the directive's function when read from the config
file                            instead of when walking the tree.
```

The fifth argument is the type of argument the directive accepts. This is one of the following. I have put the correct prototype for the function underneath each function type.

```
RAW_ARGS  --   cmd_func parses command line itself
       const char *(*raw_args) (cmd_parms *parms, void *mconfig,
                                const char *args);
TAKE1     --   one argument only
       const char *(*take1) (cmd_parms *parms, void *mconfig, const char *w);
TAKE2     --   two arguments only
       const char *(*take2) (cmd_parms *parms, void *mconfig, const char *w,
                                const char *w2);
ITERATE   --   one argument, occuring multiple times (e.g., IndexIgnore)
       const char *(*take1) (cmd_parms *parms, void *mconfig, const char *w);
ITERATE2  --   two arguments, 2nd occurs multiple times (e.g., AddIcon)
       const char *(*take2) (cmd_parms *parms, void *mconfig, const char *w,
                                const char *w2);
FLAG      --   One of 'On' or 'Off'
       const char *(*flag) (cmd_parms *parms, void *mconfig, int on);
NO_ARGS   --   No args at all, e.g. </Directory>
       const char *(*no_args) (cmd_parms *parms, void *mconfig);
TAKE12    --   one or two arguments
       const char *(*take2) (cmd_parms *parms, void *mconfig, const char *w,
                                const char *w2);
TAKE3     --   three arguments only
       const char *(*take3) (cmd_parms *parms, void *mconfig, const char *w,
                                const char *w2, const char *w3);
TAKE23    --   two or three arguments
       const char *(*take3) (cmd_parms *parms, void *mconfig, const char *w,
                                const char *w2, const char *w3);
TAKE123   --   one, two or three arguments
       const char *(*take3) (cmd_parms *parms, void *mconfig, const char *w,
                                const char *w2, const char *w3);
TAKE13    --   one or three arguments
       const char *(*take3) (cmd_parms *parms, void *mconfig, const char *w,
                                const char *w2, const char *w3);
```

The final field in the command_rec is a description of the directive. This will be output when Apache is started with the -l option.

Handler Table

The next field in the module structure is the handler table. This is a table of handler_recs, and allows modules to generate data. The handler_rec has two fields. The first is a character string that the handler is referred to by the server itself. This name is used by mod_mime's Add/Remove Handler directives. The second field is the function to use when this modules handler should be used. The prototype for this functions is:

```
static int example_handler(request_rec *r);
```

The request_rec that is passed in is the current request. The return value from this function should be either OK, DECLINED, or an HTTP error code. Only the first handler to return a value other than DECLINED is called for a request. A return value of OK means that this handler has inspected the request and has taken care of serving it. A return value of DECLINED means that the handler has inspected the request and determined that this handler is not supposed to be called for this request. Finally, an HTTP error code signifies that an error page should be returned to the client.

Register Hooks

The final field in the module structure is the register hooks function. This is function is used to replace all of the fields that were removed from the 1.3 module structure. The prototype for this function is:

```
static void example_register_hooks(void);
```

Inside of this function the module can add its functions to the list of functions called for each hook. This is done through the use of ap_hook functions.

AP_HOOK_* functions

There are a set of functions that insert a function into the list of functions called for each hook. Each of these functions has the same basic prototype, although the details differ. The basic prototype for these functions is:

```
ap_hook_foo(function pointer, char **Predecessors, char ** successors, int
where);
```

The name of the function that is called determines which hook the module's function is inserted into. For example, ap_hook_read_request(...) inserts a function into the read_request hook. The first argument is the function to insert. The prototype for those functions differ based on the hook, and we will cover the current hooks later in this paper. The second and third arguments allow for very fine grained control over where this function is added. They are both arrays of character strings. The strings in the array should be the name of other Apache modules. The second argument is a list of modules that modules whose function must be run before this module, the third is a list of modules whose functions must be run after. For example, in mod_userdir.c:

```
static void register_hooks(void)
{
    static const char * const aszSucc[]={ "mod_alias.c",NULL };

    ap_hook_translate_name(translate_userdir,NULL,aszSucc,AP_HOOK_MIDDLE);
}
```

This specifies that the translate name function for mod_alias must be called before mod_userdir's translate name function. It is important to realize that the successor and predecessor lists are only for the function that is currently being registered. The final argument to the ap_hook function is the general position for the function. This is not a fine-grained control, but it is still useful. The majority of modules will use AP_HOOK_MIDDLE for this argument. This specifies that it doesn't matter where the function is inserted in the list. It is also possible to specify AP_HOOK_FIRST, AP_HOOK_LAST, AP_HOOK_REALLY_FIRST, and AP_HOOK_REALLY_LAST. The first two state that the function should be inserted towards the front or end of the function list, but they say nothing about where it will be in relation to other modules that use the same value. The last two values provide a way to say "this needs to be the absolute first or last function called for this hook." If there are two or more functions registered with this value, nothing can be determined about which will really be called first.

Module Hooks

The first hook available for modules to use is the pre_config hook. This provides modules an opportunity to be called before the server has read the configuration hook. The prototype for this function is:

```
static void pre_config(apr_pool_t *p, apr_pool_t *ptemp, apr_pool_t *plog);
```

The first pool is a pool to be used for general allocation. The second is a log pool. This pool is cleared after each reading of the configuration file. The final pool is a temporary pool, and it is cleared often.

The next hook available for modules to use is the post_config hook. This provides modules an opportunity to be called after the server has read the configuration hook. The prototype for this function is:

```
static void post_config(apr_pool_t *p, apr_pool_t *ptemp, apr_pool_t *plog,
                        server_rec *s);
```

The pools passed to the post_config hook mirror those passed to the pre_config hook. The biggest difference is the addition of the server_rec, which is the default server for this instance of Apache.

After the post_config hook is the open_logs hook. This provides modules an opportunity to open any logs files that they will need while running. The prototype is the same as the post_config hook.

The final hook called during server setup is the child_init hook. This hook is called everytime a child process is started just after the child process has changed it's effective user ad group ids. The prototype for this function is:

```
void child_init_hook(apr_pool_t *pchild, server_rec *s);
```

The first argument is a pool that is valid for the lifetime of the child process. The second argument is the default server_rec for the current instance of Apache. All of the hooks discussed above run all of the registered functions.

The second group of hook functions are called while Apache is processing a request. All of these hooks have the same prototype:

```
int fuction_name(request_rec *r);
```

The request_rec passed in is the current request. Each of these functions has a different purpose. Some of these hooks call all functions regardless of the return value from previous functions. Others run until a function returns something other than declined.

The first function in this group is the post_read_request function. All post_read_request functions are always run. This is the first function called after the request has been read from the client.

Following post_read_request is the translate_name hook. This function gives modules a chance to translate a URI into a filename. The functions are run until one returns something other than DECLINED.

The next function is the header_parser function. This gives modules a chance to look at the headers and take appropriate action early in the request processing. All header_parser functions are always run.

The next function is the access_checker. This routine allows modules to check for module-specific restrictions upon the request resource. All access_checker functions are always run.

The next function is the check_user_id function. This provides a means for modules to ensure that the provided username and password is a valid combination for this resource. Check_user_id functions are called until one returns something other than DECLINED.

Following check_user_id is the auth_checker function. The auth checker should be used to ensure that the request has satisfied all of the requirements for the request. Auth_checker functions are called until one returns something other than DECLINED.

The next hook is the type_checker. This function is used to determine and/or set the various document type information for the currecnt request. Type_checker functions are run until a function returns something other than DECLINED.

The next function is the fixups function. This gives modules an opportunity to modify the request headers just before the content generator or handler is called. All fixups functions are always called.

The final function called is the log transaction function. After the request has been served, all logging functions are called, so that modules can write out logs about this request.

Filtered I/O

There is one final topic for writing Apache 2.0 modules - filters. Filters have recently been added to Apache, and they allow one module to modify the data generated by another module. This feature is still under development, so I am going to touch on it briefly in this paper, and will have more information at ApacheCon. The first thing is another function that is usally called from the register_hooks function. This is ap_register_filter. The prototype for this functions is:

```
ap_register_filter("CORE", core_filter, AP_FTYPE_CONNECTION);
```

The first argument is the name of the filter to register, the second is the filter fuction, and the final argument is the filter type. The filter type is either AP_FTYPE_CONNECTION, meaning that the filter is connection based and does not operate on the content itself, or AP_FTYPE_CONTENT, which means that the filter is a content filter, and it will be modifying the content.

Filter functions should have the prototype:

```
static int core_filter(ap_filter_t *f, ap_bucket_brigade *b);
```

The first argument is the current filter that is being called. This allows filters to save data for the next time they are called. The second is the data being filtered. A bucket brigade is a list of data chunks. The data is kept in buckets which all have read functions. By calling those read functions, filters can access the data to either modify or remove it from the current response.

To insert a filter into the filter chain, modules should call ap_add_filter, which takes a filter name, a pointer to a structure that the filter can use to save information and the request to associate the filter with.

Filters pass data to the next filter in the form of bucket brigades, by calling ap_pass_brigade. Ap_pass_brigade takes the next filter and the brigade to pass as arguments.

Currently, the best way to learn to write a filter is to read the core filter in http_core.c. By the time ApacheCon rolls around, there we will have more experience writing filters, and I will have more information.